RL-TR-97-123
In-House Report
October 1997

EVALUATION OF THE LARCH/VHDL INTERACTIVE PROVER IN HARDWARE VERIFICATION

Robert J. Paragi, Michael P. Nassif and Edward P. Stabler

Rome Laboratory
Air Force Materiel Command
Rome, New York

RL-TR-97-123 has been reviewed and is approved for publication.

APPROVED: Eugene C. Blackburn

EUGENE C. BLACKBURN
Chief, Electronics Reliability Division
Electromagnetics & Reliability Directorate

FOR THE DIRECTOR: John J. Bart

JOHN J. BART, Chief Scientist
Reliability Sciences

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE October 1997	3. REPORT TYPE AND DATES COVERED In-House

4. TITLE AND SUBTITLE EVALUATION OF THE LARCH/VHDL INTERACTIVE PROVER IN HARDWARE VERIFICATION	5. FUNDING NUMBERS PE - 62702F PR - 2338 TA - 01 WU -8V
6. AUTHOR(S) Robert J. Paragi, Michael P. Nassif, and *Edward P. Stabler	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rome Laboratory/ERDD 525 Brooks Road Rome, NY 13441-4505	8. PERFORMING ORGANIZATION REPORT NUMBER RL-TR-97-123
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/ERDD 525 Brooks Road Rome, NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-97-123

11. SUPPLEMENTARY NOTES

* Dept of ECE, Syracuse University. Rome Laboratory Project Engineer: Robert J. Paragi/ERDD/315-330-3547

12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report concludes an in-house evaluation of the Larch/VHDL hardware design verification tool. The evaluation is part of a larger activity to transition Larch/VHDL from a research phase to application usage within universities and industry. The Larch/VHDL tool environment has been developed by Odyssey Research Associates (ORA) under a contract with Rome Laboratory that combines a specification language, Larch, with a widely used hardware design language, VHSIC Hardware Description Language (VHDL). These two notations provide a highly structured input to the third major component of the tool environment, the Penelope theorem prover, also developed by ORA under Rome Laboratory contract. In conjunction with traditional hardware design simulation, the theorem prover provides a compact methodology for verifying correctness of a design which otherwise whould be computationally unfeasible with simulation alone. The evaluation has shown that significant portions of completed verification work on one portion of a design can be reused for proving correctness of other portions of the design.

14. SUBJECT TERMS VHSIC Hardware Description Language (VHDL)	15. NUMBER OF PAGES 48
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Table of Figures

Table of Listings

Table of Tables

Introduction

This technical report summarizes an evaluation of the Larch/VHDL hardware design verification tool environment. The tool evaluation is based upon the formal verification of the Multi-Technology Processor (MTP) design.

A Larch/VHDL tool environment has been developed by Odyssey Research Associates (ORA) under contract with Rome Laboratory. The tool combines a specification language, Larch [2], with a widely used hardware design language, VHSIC Hardware Description Language (VHDL) [14]. These two notations provide a highly structured input to the third major component of the tool environment, the Penelope theorem prover, also developed by ORA under contract with Rome Laboratory. The purpose of this in-house effort is to evaluate the Larch/VHDL hardware design verification tool for usability and soundness. The evaluation is part of a larger activity to transition Larch/VHDL from a research phase to application usage within universities and industry. The VHDL application model used for the evaluation is the Multiple Technology Processor (MTP), a 32-bit integer arithmetic and logic unit (ALU) with a register file. (See Appendix B.) Verification of the MTP included checking ALU results and checking register flag settings for arithmetic overflow, zero-result, and conditional instruction execution. Results of symbolically executing many instructions on the MTP model have been verified as correct. Further, detailed properties of instruction execution (correct ALU results for arithmetic and logical operations, plus correct flag settings for overflow, conditional operation results, and zero-result) have been verified against a VHDL register transfer level model of the MTP. Of the 26 MTP instructions, 21 have been verified to this level.

Background

The Larch/VHDL verification environment is a combination of Larch, VHDL, and the Penelope theorem prover. Larch is a two-tiered specification language developed at the Massachusetts Institute of Technology (MIT). The first tier, the Larch Shared Language (LSL) [4], is a first order predicate calculus used to build the traits, or theories, that define the sorts (VHDL types) used by the target language, i.e., bit, word, string, arrays, integer, function, etc. The second tier, called the interface language [2], defines the communication mechanisms of the target language, Ada, C, C++ or in this case VHDL, in the Larch notation. LSL is used to mathematically model data objects and operations on those objects, while the interface language maps the VHDL model into the abstractions represented by the Larch expressions for the purpose of formal reasoning.

VHDL evolved in part from a subset of the general purpose programming language Ada, and has been extended with capabilities to describe complex timing situations in hardware designs [14].

Larch/VHDL is an interactive environment that helps its user to develop and verify digital electronic hardware designs written in VHDL. Larch/VHDL is well suited to developing code in the goal-directed style advocated by Gries [3] and Dijkstra [1]. In this style the designer develops a VHDL model from a specification in a way that assures the VHDL model will meet the specification.

In order to verify a large design, we will want to decompose the design into manageable pieces (even if we do not want to, we will be forced to) that we can specify and verify separately. These pieces will need to be combined to form the full design and the way the pieces interact will have to be rigidly defined to enable us to reason about the combined system. To be useful, this decomposition process should be fully hierarchical so that the pieces can be further decomposed. To do this kind of decomposition in a disciplined way we will therefore need all the machinery that is provided by the entities, ports, component instantiations, port maps, etc. provided by a hardware description language like VHDL. To reason about how the state of the design evolves in time, we will want to reason about when individual pieces of data change and how long they have been stable, so we need the equivalents of the signals, events, stable, delayed, etc. provided by VHDL. Typical mathematical structuring constructs such as parameterized Larch traits or parameterized theories in PVS (a theorem prover generally used to verify consistency of requirements specifications) are not directly applicable to such needs.

In short, since all the machinery of an HDL is needed to verify a large design, it makes sense to use an existing well designed language like VHDL to organize the verification effort rather than reinvent such machinery inside a mathematical language.

Of course, we can use verified designs written in VHDL to communicate the design to other tools for simulation, synthesis, and layout. This is a beneficial feature of an HDL based verification system but it is not the main reason for choosing this approach. The main goal is specification and verification of designs independent of how they are described, but we claim that the structure provided by the HDL is essential to achieving this goal, so we might as well make use of the hardware description language for these other tasks as well.

The advantage of starting with a specification (in this case written in Larch) is that the specification is precise and compact. Also, in a hierarchical design process, the higher level designs may be refined to specifications of components. This allows for each refinement to be traced back to the original specification for compliance, i.e. verification. The integration of Larch and VHDL provides a path from specification to implementation.

The Larch/VHDL environment includes a large body of traits that define the basic constructs of digital design such as bit, vector, gate, logic operations and so on. Traits define sorts (logical types, including functions) and state properties or assertions that must hold true. Traits also contain theorems which are statements that are deducible from assertions, previously deduced theorems, and/or the assertions or theorems of other traits that are included. The two-tiered Larch approach allows designers the capability to extend the library of traits in order to support user defined sorts in their models. Once implemented, the traits are available as library components for reuse in other applications.

Traits are used to capture the concepts and relationships used in digital design, such as arithmetic, arrays, and lists. To support VHDL semantics there are traits defining signals, and signal delay, and other concepts needed to express the semantics of VHDL. Traits also describe the relationship between bit level operations and their arithmetic interpretation, in twos-complement or unsigned bit-level representations.

The Penelope theorem prover was originally developed for proving the correctness of Ada procedures. It is the purpose of the Penelope theorem prover to assist the user in proving theorems from the supplied axioms and included traits by applying logic reduction rules according to user directions and indicates to the user what, if anything, still has to be proved after each step. Penelope includes a simple proof editor/checker for predicate calculus that provides a number of proof rules for performing simplification and proofs.

ORA would like Penelope to automatically simplify the logic conditions that it computes, and to automatically prove the verification conditions if possible. Unfortunately, all but the most trivial simplification and proofs in Penelope require the guidance and control of the user. This interaction is necessary because of the well-known fact that simplification and theorem proving are in general undecidable; even so called automatic theorem provers usually require a good deal of guidance from human beings.

The motivation for developing this type of verification tool environment is that it is infeasible to exhaustively simulate all possible combinations of inputs to hardware designs because the design complexity and data path widths have increased. Our goal was to develop a formal verification tool that complements the current (simulation-based) hardware design process.

The Larch/VHDL verification process augments the design process in four ways:

1. By developing Larch specifications which are independent of technology and implementation details:

   A. Specifications are unambiguous, concise and immune to errors in translation from one natural language to another.

   B. Specifications may be proven to be correct.

   C. Specifications may be combined to form new specifications.

   D. Specifications may be implemented in hardware or software.

2. By verifying the correctness of a VHDL model.

Determining the correctness of a design by exhaustive simulation is not a feasible methodology due to the cost and time required to generate the test vector set and simulate the model. Formal verification of a hardware design can increase the designer's confidence that a digital circuit satisfies certain properties by reasoning over every possible input condition.

3. By verifying multiple implementations of a design in VHDL.

   In many cases, several VHDL architectures may be developed in order to perform trade-off analysis. While multiple architectures will conform to the same entity interface, their hardware implementations will vary in cost, area or performance.

4. By supporting hierarchical verification.

   The Larch/VHDL methodology supports a form of verification of single components or cells of a design which is done earlier in the development cycle than simulation is typically done. Once verified, these components are available for reuse in other designs.

Figure 1 shows the relations among VHDL designs and Larch specifications in the design hierarchy. The figure is separated vertically into three levels. The topmost level represents the final product of the design activity. Proceeding from the highest level of abstraction to lowest, boxes on the left side of the figure represent VHDL entity/architecture pairs, and a design's hierarchical decomposition. This path represents the actual design's hierarchy. The right side of the figure tracks the left side's development, augmented with Larch specifications. At each level the entity establishes transfer of information between VHDL architecture (implementation) and specification (abstraction) by specifying the set of variables used to convey this information. This junction is emphasized again in Figure 6, in the context of describing a graphically-assisted design and verification process within Larch/VHDL.

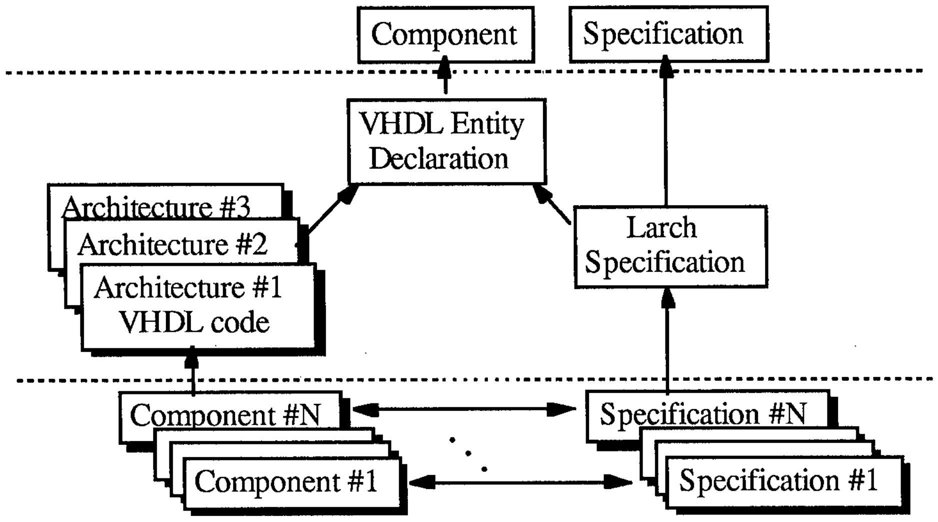

Figure 1 Design Environment

Combining Specifications and Proofs

We turn now to the process of verifying VHDL designs. Figure 2 shows the verification process, and relates the user supplied components, a VHDL design and its Larch specification, and tool supplied theories to the process of performing a proof to verify correctness of the design. On the top right of Figure 2 are VHDL designs and the associated Larch specifications. A set of tool supplied theories (the top left of Figure 2) defines design attributes, such as bit vectors, that have a purpose common to all designs.

ORA has developed a formal semantics of VHDL composed of definitions of VHDL constructs in the Larch notation, such as timing and state. Larch theories (also called "traits") are essentially self-contained sets of theorems that form a foundation in the general mathematics of sets, elementary constructs in the integers, bit vector manipulations, and conversion of bit vectors to integers for twos-complement integer arithmetic with truncation. These theories are organized as libraries and form the foundation for reasoning about designs.

The total collection of Larch theories is far more extensive than was needed for this in-house effort. Other Larch theories available cover specifications for hardware designs in VHDL and several areas of mathematics useful in general purpose logic and computer science.

Larch/VHDL uses the VHDL semantics, Larch theories, and VHDL design information (entity and architecture) as input and produces the verification conditions ("VC's", shown in the top center of Figure 2) as output for the user to prove. The VC's must be proved to be correct using the Penelope theorem prover.

The actual proof process is iterative. If the proof attempt is not successful the user modifies the specification or the VHDL architecture, and the VC's are automatically regenerated. The user begins again to interactively prove the revised VC's.

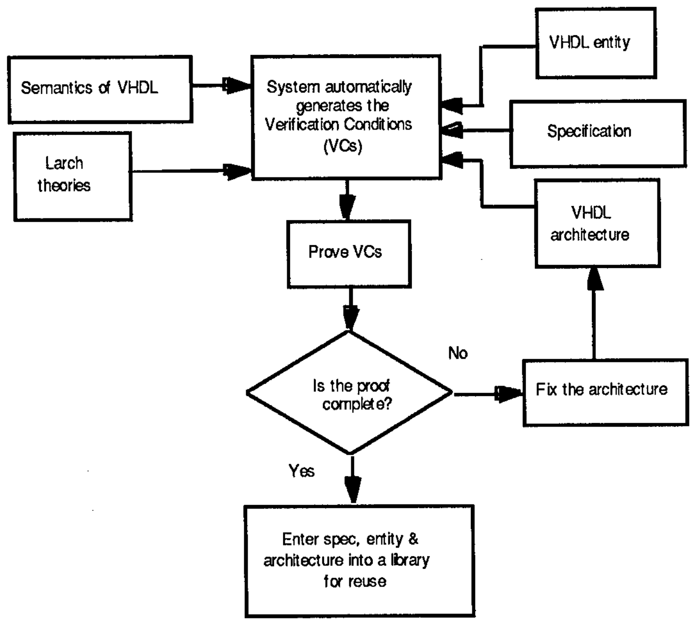

Figure 2 The Larch/VHDL verification process

Applications

The focus of most of the activity in evaluating the Larch/VHDL tool is the MTP design. It has been modeled at two levels, one level fairly abstract (behavior level) and the other more concrete (register transfer level). Details on these two levels are given below following the section "Terminology and Example Design" below.

Verification at Multiple Levels

Motivation for proving design correctness at multiple levels derives from the complexity of even a simple device design. This complexity is dealt with first by describing the device's architecture from the perspective of someone programming the device with assembly language, a perspective somewhat removed from the details of ALU operation and data paths. The assembly language view verification of correctness is performed within this model and is intended to find conceptual errors early in the design process. A more detailed view at a lower level is provided from the hardware designer's perspective. At this level errors which can affect the operation of a smaller part of the design can be found.

Terminology and Example Design

The following discussion is focused on digital logic, whether considered as an expression of a device's purpose or as a collection of logic gates, "the design", while a VHDL description of a design is known as a "model".

Properties of a model of a digital logic hardware design can be expressed at more than one level of detail: Instruction Set Architecture (ISA), behavioral, or Register Transfer Level (RTL). "Behavioral" design properties are those that are typically generated from the requirements of the intended user of the system. Behavioral properties generally are informal, and state functional requirements in natural language. The requirements are then translated into the language of some formal specification system. The process from here forward is one of trying to match a requirements property with one or more properties of the model to determine if the requirement is met within the model. The process has often been described as checking to see if the implementation (model) meets (implies) the specification (requirements).

Applying the Penelope Theorem Prover

In the context of a VHDL design, the original proof obligation is the Verification Condition (VC) (Figure 2), which is generated automatically by the Larch/VHDL tool. During the proof process the original proof property (VC) considered as an obligation is converted by each proof step into another proof obligation. A proof is complete when there is no remaining proof obligation. The proof ends with "BY synthesis of TRUE" or "BY analysis of FALSE".

In most sequences of proof steps assumptions are involved, and they are designated "hypotheses". The "conclusion" (symbolically, "hypotheses" -> "conclusion") based on these hypotheses is a proof obligation. Formally, the conjunction ("anding" together) of the hypotheses implies the conclusion.

A property could be an axiom, a basic assumption, usually about some physical situation, where following the keyword "asserts" are axioms about writing and reading memories, plus an axiom particular to the register file of a design described below.

A VHDL entity defines the input and output signals for a design, and exists as a file apart from the VHDL architecture that contains design implementation details. The physical separation of entity and architecture allows association of multiple architectures to a single entity. Specifications relate to VHDL directly at the entity declaration by referencing variables listed in the entity interface file. In this view creating the entity is the beginning of the formal design process, in that both the Larch-like specification and the VHDL architecture follow from it. Just such a chronology is followed with graphical assistance to the process of creating files and constructing proofs, as described below.

The MTP

The application used for evaluation of the Larch/VHDL hardware verification environment is a VHDL model of the Multiple Technology Processor (MTP), a 32-bit integer arithmetic and logic unit (ALU) with a register file and associated datapaths all on a single chip. A block diagram for the MTP is shown in Figure 3 below. See Appendix B also.

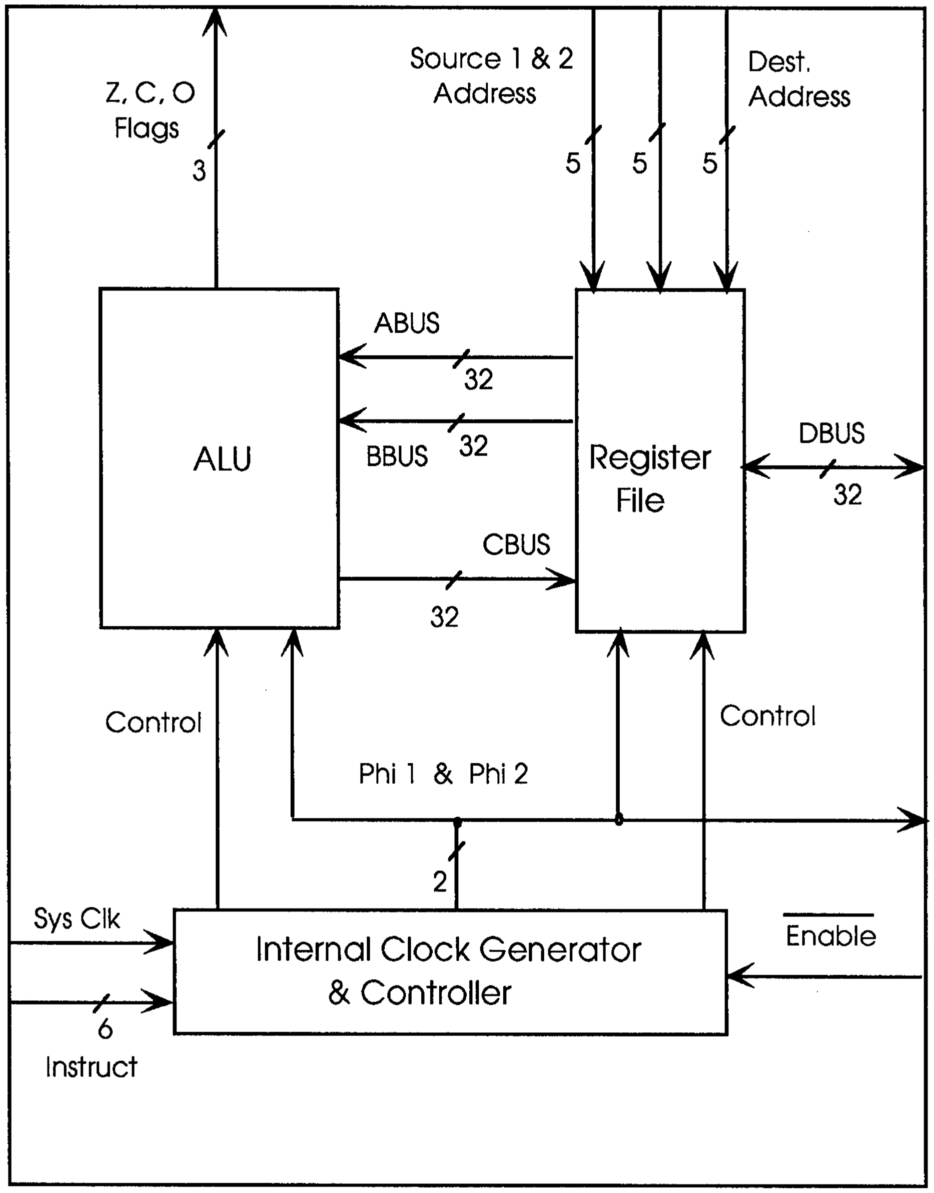

Figure 3 MTP Block Diagram

The word "specification" is used often throughout this report in the context of the MTP example, and refers to the formal logic descriptions of the MTP as developed from the informal descriptions in English text, written in the Larch specification language. Our approach is to describe and reason about the design at the level of integer arithmetic, which is more intuitive to the user. Integer arithmetic is supported by twos-complement arithmetic properties (subtraction implemented as addition), and to using bit-string ("bit-vector") representations where bit-level operations are necessary. The use of bit-string representations further requires reasoning about truncation of these strings to a fixed width, as required by the precision of the targeted ALU and the width of data paths in the design. In the actual proof reasoning about truncation and extreme values of integers (overflow and underflow) invariably reduces to simplifying expressions containing powers of two. Then the problem of completing the proof is back in the realm of integer arithmetic.

Instruction Set Architecture Property Proofs

One "behavioral" view of the design is that of an Instruction Set Architecture (ISA). The ISA views the design's function in terms of the assembly language instructions which execute on the hardware. This is a level high enough that capabilities identified in the requirements for the design can be distinguished (one instruction for each capability), but still primitive enough that each instruction could be implemented in a small microcode function. The inspiration for proving properties of the design at the ISA level came from a Syracuse University project [16] to model and reason about the AMD 2910 controller microcircuit.

The ISA model expresses input/output properties of ALU results, including register file updates with the results, and properties of flags associated with ALU operations. As shown in the example below, a typical relation would involve an instruction to be executed and data as inputs, and an updated register file, updated flags, and more data as outputs. The relation provides the connection between the inputs and the outputs.

Listing 1 is a specification of the ISA model. A call to the ALU functionality ("alu" function invoked from the last function, "exe") and updating three flags are the organizational focus of this specification. The updating itself is accomplished with iterations of "if...then...else" clauses, one clause for each instruction. The "alu" function is defined in a subsidiary file of specification. It is implemented with iterations of "if...then...else" clauses also, and is explained in more detail below. The theorems of this specification are created by substituting into the "exe" function the specific values particular to each instruction. The function "exe" itself implements execution of an instruction. Its parameter list includes both current and updated values of the register file ("rf") and flags. Terminology associated with this file is expanded upon further below. The declarative style of this specification is evident in the appearance of both current and updated values in the "exe" function. The effect of this function is communicated to the caller in changes made between current and updated values of parameters.

```
--| library lib
--|         INITIAL_THEORY
--|         GENERAL_MATH
--|         STD
--|         WORK
--|         VHDL_MATH ;
--! Verification status: Verified
 ---> trait MTP has unfinished proofs
--| Larch
MTP: trait
  includes (new_MTP_syntax, Int2BV, BitVector, Bit_vector_to_int)
  includes ({Int}Vector)(Int)
includes ({Bit}Vector)(Bit)
introduces
 exe: Inputs, Outputs -> Bool
 maxint, minint: -> Int
 up_o_flags: Inst, Int, Int, Int, Int, Vector[Int], Bool -> Bool
 up_z_flags: Inst, Int, Int, Int, Int, Vector[Int], Bool -> Bool
 up_c_flags: Inst, Int, Int, Int, Int, Vector[Int], Bool -> Bool
 exe: Inst,
  Int,
  Int,
  Int,
  Int,
  Vector[Int],
  Bool,
  Bool,
  Bool,
  Vector[Int],
  Bool,
  Bool,
  Bool,
  Int -> Bool

asserts
 forall vec, nrf, rf:Vector[Int], i:Inst,
 n, dr, sr1, sr2, dbusin, dbusout:Int, o_in, z_in, c_in, o, z, c:Bool
  maxi: maxint() = 2147483647
  regendian: rf'ascending = false
  regleft: rf'left = 31
  regright: rf'right = 0
  nregendian: nrf'ascending = false
  nregleft: nrf'left = 31
  nregright: nrf'right = 0
  flag_op_o: up_o_flags(i, sr1, sr2, dr, dbusin, rf, o)
    =
    (if ASHL() = i
     then not (i2bv(rf[sr1], ev)[i2bv(rf[sr1], ev)'left]
        =
        i2bv(rf[sr1], ev)[i2bv(rf[sr1], ev)'left-1])
     else (if ((ASHR() = i or SHR() = i) or SHL() = i) or MV() = i
        then false
        else (if ADD() = i
           then rf[sr1]+rf[sr2]<(-1)*maxint()-1
             or
             maxint()<rf[sr1]+rf[sr2]
           else (if SUB() = i
             then rf[sr1]-rf[sr2]<(-1)*maxint()-1
```

```
                  or
                   maxint()<rf[sr1]-rf[sr2]
                else (if INCR() = i
                   then rf[sr1]+1<(-1)*maxint()-1
                      or
                       maxint()<rf[sr1]+1
                   else (if DECR() = i
                      then rf[sr1]-1<(-1)*maxint()-1
                         or
                          maxint()<rf[sr1]-1
                      else (if NEG() = i
                         then (-1)*rf[sr1]<(-1)*maxint()-1
                            or
                             maxint()<(-1)*rf[sr1]
                         else o)))))))
flag_op_z: up_z_flags(i, sr1, sr2, dr, dbusin, rf, z)
  =
   (if (((ASHL() = i or ASHR() = i) or SHL() = i) or SHR() = i)
     or
      MV() = i
    then 0 = rf[sr1]
    else (if ADD() = i
       then 0 = rf[sr1]+rf[sr2]
       else (if SUB() = i
          then 0 = rf[sr1]-rf[sr2]
          else (if INCR() = i
             then 0 = rf[sr1]+1
             else (if DECR() = i
                then 0 = rf[sr1]-1
                else (if NEG() = i
                   then 0 = (-1)*rf[sr1]
                   else (if AND_I() = i
                      then 0
                        =
                         int(i2bv(rf[sr1], ev)
                           and
                            i2bv(rf[sr2], ev))
                      else (if NAND_I() = i
                         then 0
                           =
                            int(i2bv(rf[sr1], ev)
                              nand
                               i2bv(rf[sr2], ev))
                         else (if NOR_I() = i
                            then 0
                              =
                               int(i2bv(rf[sr1], ev)
                                 nor
                                  i2bv(rf[sr2], ev))
                             else (if OR_I() = i
                               then 0
                                 =
                                  int(i2bv(rf[sr1], ev)
                                    or
                                     i2bv(rf[sr2], ev))
                               else (if XOR_I() = i
                                 then 0
                                   =
                                    int(i2bv(rf[sr1], ev)
                                      xor
                                       i2bv(rf[sr2], ev))
                                 else (if NOT_I() = i
                                   then 0
                                     =
                                      int(not i2bv(rf
                                          [sr1], ev))
                                   else z))))))))))))
flag_op_c: up_c_flags(i, sr1, sr2, dr, dbusin, rf, c)
  =
   (if conditional(i)
     then (if EQ() = i
        then rf[sr1] = rf[sr2]
        else (if GT() = i
          then rf[sr1]>rf[sr2]
          else (if GTE() = i
            then rf[sr1]>=rf[sr2]
            else (if LT() = i
              then rf[sr1]<rf[sr2]
              else (if LTE() = i
                then rf[sr1]<=rf[sr2]
                else (if NEQ() = i
                  then rf[sr1]/=rf[sr2]
                  else c))))))
     else c)
 meet: exe(i, sr1, sr2, dr, dbusin, rf, o_in, z_in, c_in, nrf, o, z, c,
     dbusout)
   =
   ((in_domain(sr1, rf) and in_domain(sr2, rf)) and in_domain(dr, rf)
   ->
    ((nrf = rf[dr=>alu(i, sr1, sr2, dr, rf)]
        and
         o = up_o_flags(i, sr1, sr2, dr, dbusin, rf, o_in))
```

```
            and
             z = up_z_flags(i, sr1, sr2, dr, dbusin, rf, z_in))
           and
            c = up_c_flags(i, sr1, sr2, dr, dbusin, rf, c_in))

implies
  forall a, b, sr1, sr2, dr, dbusin, dbusout:Int, o, z, c:Bool,
  nrf, rf:Vector[Int]
    eq_reg: a = b->rf[dr=>a] = rf[dr=>b]
    same_rf: rf[dr=>rf[dr]] = rf
    meet1 (rewrite) : exe(ADD(), sr1, sr2, dr, dbusin, rf, o, z, c,
       rf[dr=>rf[sr1]+rf[sr2]],
       rf[sr1]+rf[sr2]<(-1)*maxint()-1 or maxint()<rf[sr1]+rf[sr2],
       0 = rf[sr1]+rf[sr2], c, dbusin)
    meet2 (rewrite) : exe(EQ(), sr1, sr2, dr, dbusin, rf, o, z, c, rf, o, z,
       rf[sr1] = rf[sr2], dbusin)
    meet3 (rewrite) : exe(STO(), sr1, sr2, dr, dbusin, rf, o, z, c, rf, o,
       z, c, rf[dr])
    meet4 (rewrite) : exe(LD(), sr1, sr2, dr, dbusin, rf, o, z, c,
       rf[dr=>dbusin], o, z, c, dbusin)
    nopchk (rewrite) : exe(NOP(), sr1, sr2, dr, dbusin, rf, o, z, c, rf, o,
       z, c, dbusin)
    ashl_m (rewrite) : exe(ASHL(), sr1, sr2, dr, dbusin, rf, o, z, c,
       rf[dr=>rf[sr1]+rf[sr1]],
       not (i2bv(rf[sr1], ev)[i2bv(rf[sr1], ev)'left]
        =
         i2bv(rf[sr1], ev)[i2bv(rf[sr1], ev)'left-1]), 0 = rf[sr1]+rf[sr1],
       c, dbusin)
    ashr_m (rewrite) : exe(ASHR(), sr1, sr2, dr, dbusin, rf, o, z, c,
       rf[dr=>int(sign_ext(shift(i2bv(rf[sr1], ev), -1)))], false,
       0 = rf[sr1], c, dbusin)
    incr_m (rewrite) : exe(INCR(), sr1, sr2, dr, dbusin, rf, o, z, c,
       rf[dr=>rf[sr1]+1], rf[sr1]+1<(-1)*maxint()-1 or maxint()<rf[sr1]+1,
       0 = rf[sr1]+1, c, dbusin)
    decr_m (rewrite) : exe(DECR(), sr1, sr2, dr, dbusin, rf, o, z, c,
       rf[dr=>rf[sr1]-1], rf[sr1]-1<(-1)*maxint()-1 or maxint()<rf[sr1]-1,
       0 = rf[sr1]-1, c, dbusin)
    neg_m (rewrite) : exe(NEG(), sr1, sr2, dr, dbusin, rf, o, z, c,
       rf[dr=>(-1)*rf[sr1]],
       (-1)*rf[sr1]<(-1)*maxint()-1 or maxint()<(-1)*rf[sr1],
       0 = (-1)*rf[sr1], c, dbusin)
    shl_m (rewrite) : exe(SHL(), sr1, sr2, dr, dbusin, rf, o, z, c,
       rf[dr=>rf[sr1]+rf[sr1]], false, 0 = rf[sr1], c, dbusin)
    shr_m (rewrite) : exe(SHR(), sr1, sr2, dr, dbusin, rf, o, z, c,
       rf[dr=>rf[sr1]/2], false, 0 = rf[sr1], c, dbusin)
    sub_m (rewrite) : exe(SUB(), sr1, sr2, dr, dbusin, rf, o, z, c,
    rf[dr=>rf[sr1]-rf[sr2]],
    rf[sr1]-rf[sr2]<(-1)*maxint()-1 or maxint()<rf[sr1]-rf[sr2],
    0 = rf[sr1]-rf[sr2], c, dbusin)
  and_m (rewrite) : exe(AND_I(), sr1, sr2, dr, dbusin, rf, o, z, c,
    rf[dr=>int(i2bv(rf[sr1], ev) and i2bv(rf[sr2], ev))], o,
    0 = int(i2bv(rf[sr1], ev) and i2bv(rf[sr2], ev)), c, dbusin)
  gt_m (rewrite) : exe(GT(), sr1, sr2, dr, dbusin, rf, o, z, c, rf, o, z,
    rf[sr1]>rf[sr2], dbusin)
  gte_m (rewrite) : exe(GTE(), sr1, sr2, dr, dbusin, rf, o, z, c, rf, o,
    z, rf[sr1]>=rf[sr2], dbusin)
  lt_m (rewrite) : exe(LT(), sr1, sr2, dr, dbusin, rf, o, z, c, rf, o, z,
    rf[sr1]<rf[sr2], dbusin)
  lte_m (rewrite) : exe(LTE(), sr1, sr2, dr, dbusin, rf, o, z, c, rf, o,
    z, rf[sr1]<=rf[sr2], dbusin)
  neq_m (rewrite) : exe(NEQ(), sr1, sr2, dr, dbusin, rf, o, z, c, rf, o,
    z, rf[sr1]/=rf[sr2], dbusin)
  mv_m (rewrite) : exe(MV(), sr1, sr2, dr, dbusin, rf, o, z, c,
    rf[dr=>rf[sr1]], false, 0 = rf[sr1], c, dbusin)
  mvns_m (rewrite) : exe(MVNS(), sr1, sr2, dr, dbusin, rf, o, z, c,
    rf[dr=>rf[sr1]], o, z, c, dbusin)
```

Listing 1 Specification of the ISA model

The style of the statements is declarative. This means that properties of a model are expressed as equations of logic functions with Boolean parameters. These logic functions express properties of a model when used in equations where the individual variable name parameters are substituted with Boolean expressions of logic variables. As an example (see Table 1 below), if the updated register file is denoted by "nrf", the original state of the register file (before execution of a given instruction "i") by "rf", and the output data by a function named "alu", where its parameters are instruction "i", ALU inputs from register file locations "sr1" and "sr2", and register file destination (offset into the register file as an array) "dr" of the ALU output, then the relation as expressed by an equation is "nrf = rf[dr=>alu(i,sr1,sr2,dr,rf)]". The symbol "=>" assigns ALU output at offset "dr" into the register file "rf". The objective of the proof steps yet to come is to show that the substitutions make the equations true. In substitutions involving the instruction for addition, the equation above becomes "nrf = rf[dr=>rf[sr1]+rf[sr2]]" because the "alu" function produces as output the sum of the register file values input to the ALU. For example, to add register 7 to register 12 and store the result in register 19 (R7 + R12 => R19), the substitutions shown in Table 1 would be the appropriate variable assignments.

variable definition	values
nrf : updated register file	(state = register values after update)
rf : current state of register file	(state = register values currently)
i : instruction	add
sr1 : address of ALU left input register	register 7
sr2 : address of ALU right input register	register 12
dr : address of ALU destination register	register 19

Table 1 Variables for example at ISA level

ALU operation flags indicate a zero ALU result (FZ), arithmetic overflow/underflow (FU), and the fact that the currently executed instruction is a conditional (FC), e.g., less than, greater than, etc.

Proofs of properties at the ISA level generally involve searching through "tables" indexed by instruction name and implemented as functions in the form of if-then-else chains (see Figure 4) traversing the entire list of instruction opcodes. There is one if-then-else chain for ALU results (the "alu" function mentioned above) and a separate chain for each of the three flag update functions. At the ISA level all information for developing the expression of properties to be proved is derived from specifications written in Larch only.

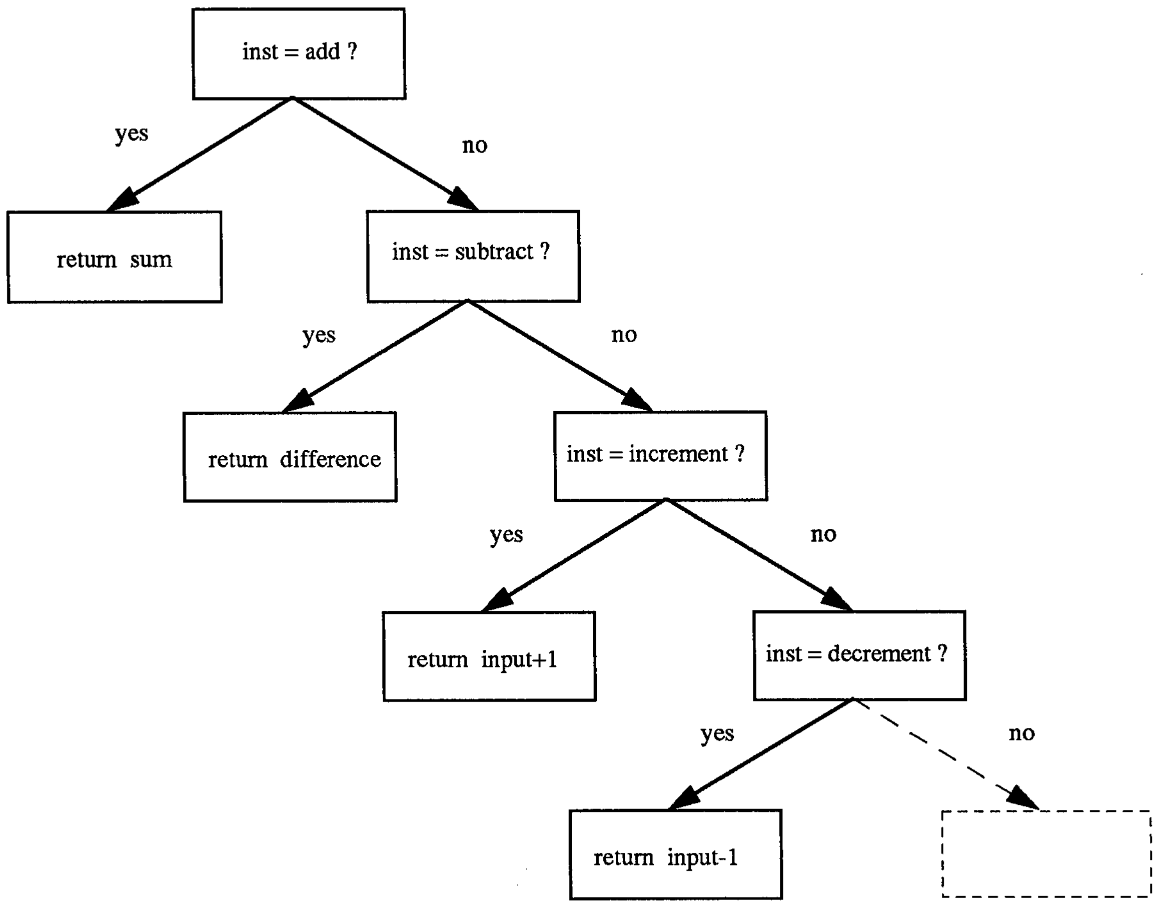

Figure 4 ALU If-Then-Else Chain

Proofs that individual instructions execute correctly under this model reflect the depth at which each instruction's actions are described. Comparison instructions and instructions for data transfer to/from external memory (not to be confused with the ALU's local register file) have very simple descriptions based on data being represented as integers. Therefore instructions in these two classifications have properties whose proofs of correctness are very simple. The proofs for most of these ISA level instructions are absolutely identical.

Register Level Property Proofs

Register level properties proved in this activity were generated from a VHDL architecture developed from the MTP model (see Appendix A), and are therefore more detailed than properties proved in the ISA view. From the MTP processor instruction set, several commonly used arithmetic, bit-logical (Boolean), and comparison instructions from the ISA level were chosen. Their properties, in addition to arithmetic and logic results, include functioning of three flags associated with these instructions: overflow flag for arithmetic functions, zero-result flag in the case of arithmetic and bit-logical instructions, and a flag indicating execution of a comparison instruction. The difference in the origin of these properties from those expressed and proved in the ISA model, as described above, is that in this case there is an additional source of information in the form of VHDL concurrent process statements. The user in this environment can write statements in Larch-like syntax about properties that are defined from information in the VHDL portion of the model.

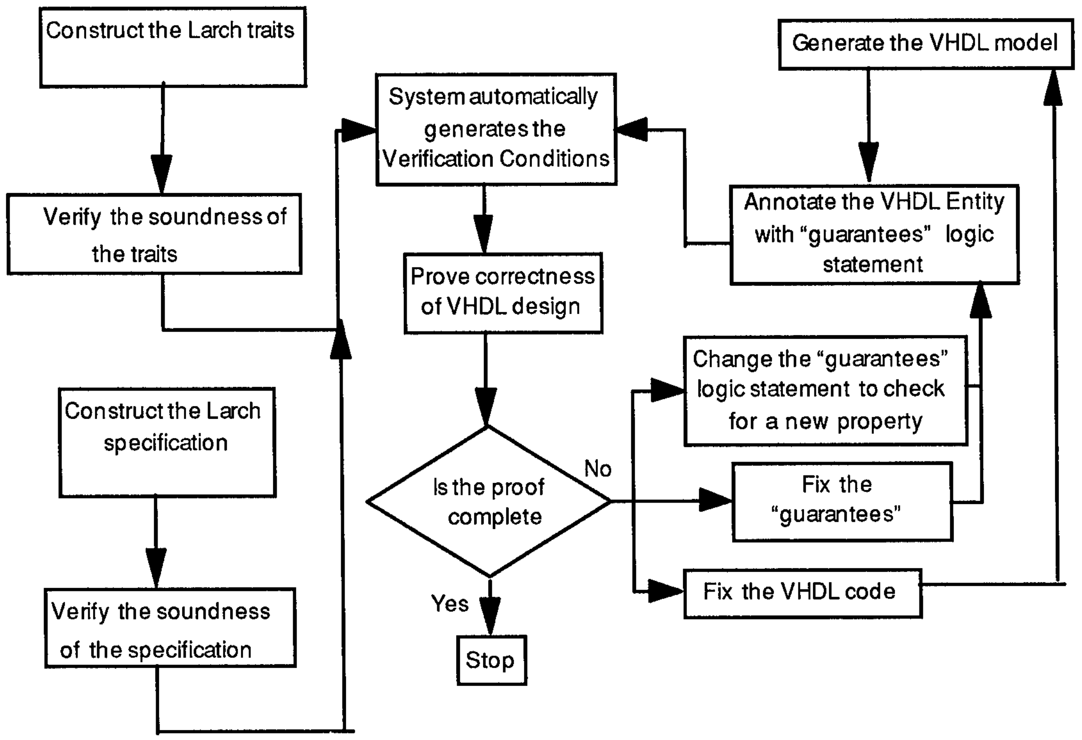
Construct the Larch traits
Verify the soundness of the traits
Construct the Larch specification
Verify the soundness of the specification
System automatically generates the Verification Conditions
Prove correctness of VHDL design
Is the proof complete
Yes
Stop
No
Generate the VHDL model
Annotate the VHDL Entity with "guarantees" logic statement
Change the "guarantees" logic statement to check for a new property
Fix the "guarantees"
Fix the VHDL code

Figure 5 Verifying VHDL Designs

With VHDL, properties to be proved can be expressed more precisely than with specifications alone. As a related benefit in an environment involving VHDL, more of the process of deriving properties to be proved (some properties described as verification conditions, or VC's) can be automated. In contrast to higher level specifications, where there is a much wider variety of syntax available for describing concepts not necessarily confined to hardware description, the majority of VHDL syntax supports very regular and repeatable constructs, such as gates, registers, and interactions among them. User supplied properties ("the specification") are also referred to as "guarantees". See Figure 5. It is a more detailed version of Figure 2 in that it emphasizes the dynamics of proof process iteration, as opposed to Figure 2, which featured inputs to this process. Much of the process is aided by organizational tools, such as a graphical user interface (GUI). Within a GUI environment the Larch/VHDL user makes choices from an associated menu to create a graph whose nodes are files used for constructing VC's. This is accomplished as follows.

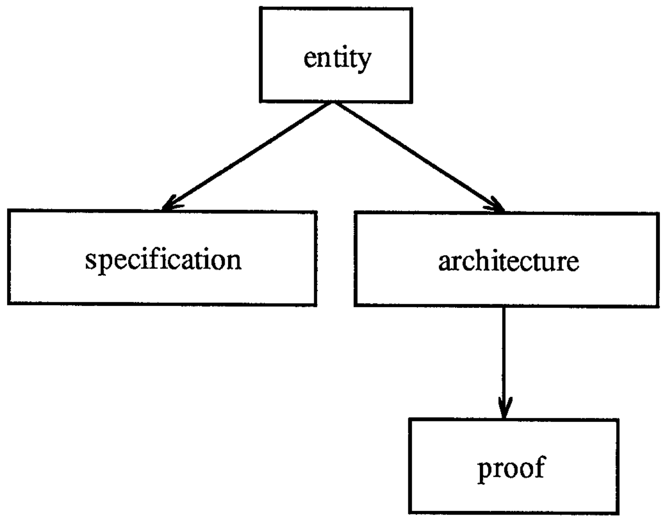

Figure 6 Graphically-Assisted Design Process

The user supplies file names and directory pathnames for four files: a VHDL entity, a VHDL architecture, proof node, and a Larch-like specification. The entity node is constructed first, and the pull-down menu at the already-constructed entity node allows even more automated construction of the associated specification node, VHDL architecture node, and proof node. Pull-down menus at the entity node, specification node, and VHDL architecture node allow an editor (e.g., emacs, vi) to create/open these files directly from their respective nodes on the graph. The specification contains the "guarantee", or property, to be proved about the VHDL architecture. See Figure 6 and Figure 1 above. The pull-down menu at the proof node allows the user to open the interactive proof editor (Penelope) and automatically construct and load the proof.

Note that depending on how much the user has interacted with the proof process already, the proof file loaded may be the file representing progress ranging anywhere from a beginning situation just after the VC's have been generated (no proof steps have been made yet), to a completed proof. These capabilities require then that the user has first constructed a VHDL entity file, a specification file, and a VHDL architecture file. Finally the proof editor is opened and loaded automatically with the VC's and other properties (the "guarantees") recognizable from the specification file. At this point the user is ready to start proving the correctness of the VC's by using the proof editor interactively. Examples of related sets of entities, specifications, and architectures are given in Appendix E.

Conclusions about Verification at Multiple Levels

The Larch/VHDL tool supports the verification of components at different levels of the design hierarchy: ISA, behavioral and RTL, and the verification of the composition of previously verified components. Both features simplify the design process and support the designer's view of the world. The user can reason about the higher level design aspects entirely within the Larch portion of the environment, and then when comfortable with the high level, use parts of the Larch specification as "guarantees" to verify the VHDL design.

User involvement in the proof process extends to more than just the ability to select proof steps and arrange the order of applying them. The user has the option of reaching proof completion by applying domain knowledge to decrease the number of proof steps. This is a more active approach than responding passively to a proof obligation that remains after applying the most recent proof step. Domain knowledge is applied to the proof by making a claim which splits the remaining proof obligation into two parts, a new proof obligation to prove correct the statement of the claim itself with the hypotheses prior to the claim's existence, and a second part which repeats the original proof obligation and hypotheses augmented by the claim as a new hypothesis. Therefore, the claim has to be proved correct, and then it can be used as an additional hypothesis with the original proof obligation. The syntax of the claim is a statement about logic variables which appear in the original proof obligation. There is a tradeoff between the opportunity for shorter proofs and the amount of domain knowledge required of the user in order to state a claim effective at shortening the proof.

The user also needs to determine optimal substitutions when instantiating theorems (see section on terminology). The challenge of this requirement is partially lessened by libraries of specifications for general mathematics (of integers) and libraries of specifications (theories) implementing VHDL semantics and high level bit vector constructs. These have been supplied as part of the Larch/VHDL environment. The theorems from these libraries are extensive, and as would be expected, the bit vector libraries have been referenced most often during this evaluation. They are built upon the general mathematics specifications and there was rarely a need either to instantiate theorems directly from general mathematics libraries, or to prove theorems beyond those already available in the delivered libraries. Most effort was devoted to using the theorems from these libraries effectively.

Despite the demands on the user's domain knowledge and judgment in setting up a proof strategy (or borrowing and modifying one), the advantage of having this particular proof editor as part of the tool environment is the support it provides for putting together a proof strategy. The libraries of proven theorems about general mathematics, logic, and special constructs needed for hardware design reasoning, such as bit vectors and their conversions to integers, relieve the user from extensive work in building a set of theorems.

Conclusions

Adopting a formal specification and proof of correctness methodology has produced several benefits over traditional simulation based design environments.

Formulating high level Larch specifications in itself uncovered design omissions and errors in the early stages of this project, particularly with regard to behavioral definitions of integer arithmetic. Specific behavioral descriptions related to shifting operations and arithmetic overflow were clarified when discrepancies were identified in the process of writing formal specifications from ordinary English descriptions of behavior. The operative difference is that although one could precede traditional simulation with a formal specification, the formal specifications used in connection with theorem proving are linked to the theorem proving activity by the "guarantees" statements referenced earlier in this report. The specification process, while as thorough as possible for capturing a designer's intentions, when followed by standalone simulation allows another source of error to be injected via misinterpretation between specification and simulation.

There are a number of reasons why theorem proving compares favorably with simulation.

As stated briefly before, proof of each theorem about a design covers correctness of multiple design input configurations otherwise treated separately under simulation. The typical argument in favor of theorem proving versus simulation for data paths is that for any combination of 0's and 1's on a bus, with the data path width as the exponent and two as a base, there are two to that exponent possible combinations of data path values. All of these combinations must be simulated separately in lieu of proving a single theorem characterizing the setting and reading (interpreting) of the entire bit field representing those values. Such theorems would generally describe the effect of a change in one bit of data in this data field on any subdesign which uses the data path as input. Beyond this initial concern over combinatorial explosion (complexity exponential in the data path width), the data path paradigm is treated much more generally by theorem proving, as many proof environments allow the user to parameterize all theorems and proofs on the width of the data path without specifying width explicitly. Alternatively, a change in data path width requires that a simulation based check for correctness would have to be repeated from the beginning.

The repetitious nature of simulation makes human interaction less effective (monotony and propensity for errors in human review of massive displays of very similar patterns) and makes no use of the hierarchy of design possible with theorem proving. Although no claim is made that portions of that hierarchy handled best by theorem proving techniques are a total replacement for simulation, the structure of such a hierarchy is made more apparent to the user of a theorem proving environment. Subtle design flaws masked by poorly chosen hierarchical structure would be harder to detect with simulation. A secondary benefit of obvious theorem prover support for hierarchy in design is that the entire design can be more easily comprehended. Since hierarchy also ensures more modularity, parts of a given design can be more easily isolated within a theorem proving environment, for later reuse in another design. In contrast, although a simulation environment allows separating parts of design for independent simulation, the combinatorial explosion problem referenced earlier is made worse when data paths are disconnected to isolate modules in a more detailed phase of a design. The number of input and output port data paths (at the boundaries of the newly created modules) is increased, thereby increasing at least linearly the total number of unique input and output combinations to be verified. The number of input and output combinations to be checked doubles for every disconnection.

Bibliography

[1] Edsger W. Dijkstra, A Discipline of Programming. Prentice Hall, Englewood Cliffs, 1976.

[2] S. Garland, J. Guttag and J. Horning, "Debugging Larch Shared Language Specification," IEEE Trans. on Software Engineering, Vol. 16, no. 9, September 1990.

[3] David Gries, The Science of Programming. Springer-Verlag, 1981.

[4] J. Guttag, J. Horning and J. Wing, "The Larch Family of Specification Languages," IEEE Software, September 1985.

[5] J. Jamsek and M. Bickford, "Formal Verification of VHDL Models," Final Technical Report, Rome Laboratory RL-TR-94-3, March, 1994.

[6] J. Wing, "Writing Larch Interface Language Specifications," ACM Trans. on Programming Languages and Systems, Vol. 9, no.1, January, 1987.

[7] Odyssey Research Associates, "Penelope Reference Manual V3-3," TM94-0009, December, 1993.

[8] M.Nassif and R. Paragi, "The Larch/VHDL Methodology for Hardware Verification", 1996

[9] M. Bickford, "User/Training Manual for Formal Verification of VHDL Design", TM-96-0024, July, 1996

[10] M. Bickford, "Final Report for Formal Verification of VHDL Design", TM-96-0025, July, 1996

[11]] E. Stabler, M. Nassif and R. Paragi , "Extending the Design Process with Formal Verification Technology", The Spring 1996 VHDL International Users' Forum Conference, February, 1996

[12] A. Barbour, "Formal Hardware Verification Using The Larch/VHDL Theorem Prover", Final Report for Summer Faculty Research Program, August, 1996

[13] E. Stabler, M. Nassif and R. Paragi, "Verification of ASIC Designs in VHDL Using Computer-Aided Reasoning", ASIC Conference, September, 1996

[14] Z. Navabi, VHDL: Analysis and Modeling of Digital Systems, McGraw-Hill, 1993

[15] G. Nelson and D. Oppen, " Simplification by Cooperating Decision Procedures", ACM Transactions on Programming Languages and Systems, Vol. 1, No. 2, October, 1979

[16] S-K Chin, "Formal Specification and Verification of the Am2910 Sequencer", CASE Center Technical Report No. 9401, January, 1994.

Appendix A: Multiple Technology Processor (MTP)

ALU VHDL Description

The first module below is the entity and the second module is an architecture implementation, exemplifying two of the boxes in Figure 6 above.

```
--entity mtp1 is
--this is the MTP without the register file
--   generic(n: integer range 1 to 10000);
--   port ( INST: Bit_vector(5 downto 0);
--           A, B: Bit_vector(n-1 downto 0);
--           Result: out Bit_vector(n-1 downto 0);
--           FO, FZ, FC : out bit);
--end mtp1;

architecture arch of mtp1 is

component adder generic (N: integer range 1 to 10000);
port (a, b: in bit_vector(N-1 downto 0);
     ci : in bit;
     s: out bit_vector(N-1 downto 0);
     co: out Bit);
end component;

component zerovec generic(n : integer);
  port ( result: out bit_vector(n-1 downto 0));
end component;

component onevec generic(n : integer);
  port ( result : out  bit_vector(n-1 downto 0));
end component;

component ovf  generic ( n: integer range 1 to 10000);
      port ( a : bit_vector(n-1 downto 0);
            b : bit_vector(n-1 downto 0);
            s : bit_vector(n-1 downto 0);
            ci : bit;
            co : bit;
            nf : out bit;
            zf : out bit;
            ov : out bit
            );
```

```
end component;


signal co,nf,one,zer,zf : bit;
signal ZERO : bit_vector(n-1 downto 0);
signal ONES : bit_vector(n-1 downto 0);
signal CARRY_IN  : bit;
signal ALU_LEFT  : bit_vector(n-1 downto 0);
signal ALU_RIGHT : bit_vector(n-1 downto 0);
signal SUM          : bit_vector(n-1 downto 0);
signal result_local : bit_vector(n-1 downto 0);
signal COMPARE  :boolean;  -- INST is Comparison
signal ARITH       :boolean;  -- INST is Arithmetic
signal LOGIC       :boolean;  -- INST is Logical
signal fz_check    :boolean;
signal fo_ovf       : bit;
begin

L1:  adder generic map(n) port map (ALU_LEFT,ALU_RIGHT,CARRY_IN,SUM,co);
L2:  zerovec generic map(n) port map(ZERO);
L3:  onevec  generic map(n) port map(ONES);
L4:  ovf generic map (n) port map(ALU_LEFT,
ALU_RIGHT,SUM,CARRY_IN,co,nf,zf,fo_ovf);

one <= '1';
zer <= '0';

COMPARE <=
        (INST = "010010") or
        (INST = "010011") or
        (INST = "010100") or
        (INST = "010101")  or
        (INST = "010110") or
        (INST = "010111");        --COMPARE is True for comparisonsA

LOGIC    <=
         (INST = "001100") or
         (INST = "001101") or
         (INST = "001110")  or
         (INST = "010001")  or
         (INST = "001111")   or
         (INST = "010000");

ARITH    <=
         (INST = "000011") or
         (INST = "000100") or
        (INST = "000101") or
        (INST = "000111") or
        (INST = "000110") or
        (INST = "001000") or
        (INST = "001010") or
        (INST = "001001") or
        (INST = "001011");

CARRY_IN <= one when (INST = "001011") else
        one when (INST = "001000") else
        one when (INST = "000110")else
        one when COMPARE  else
        zer;

ALU_RIGHT <=
         A                  when (INST = "000100") else
         ZERO               when (INST = "000101") else
         ZERO               when (INST = "001010")else
         ZERO               when (INST = "000110")else
         ONES               when (INST = "000111")else
         ZERO               when (INST = "001000") else
         A                  when (INST = "001001") else
         not B              when (INST = "001011")else
         ZERO               when (INST = "000000")  else
         ZERO               when (INST = "000001")  else
         ZERO               when (INST = "100001")  else
         not B              when (COMPARE)else
         B;


ALU_LEFT <=
        not A   when (INST = "001000") else
        A(n-1) & A(n-1 downto 1) when (INST = "000101") else
         zer    & A(n-1 downto 1) when (INST = "001010") else
        A;

Result_local  <= SUM            when (INST = "000000")else
        SUM                     when (INST = "000001")else --MVNS
        SUM                     when (INST = "100001")else
        SUM                     when (INST = "000011")else
        SUM                     when (INST = "000100")else
        SUM                     when (INST = "000101") else
        SUM                     when (INST = "000110")else
        SUM                     when (INST = "000111")else
        SUM                     when (INST = "001000")else
        SUM                     when (INST = "001001")else
```

```
          SUM                              when (INST = "001010")else
          SUM                              when (INST = "001011")else
          ALU_LEFT and ALU_RIGHT           when (INST = "001100")else
          not (ALU_LEFT and ALU_RIGHT)     when (INST = "001101")else
          not (ALU_LEFT or ALU_RIGHT)      when (INST = "001110")else
          ALU_LEFT or ALU_RIGHT            when (INST = "001111") else
          ALU_LEFT xor ALU_RIGHT           when (INST = "010000")else
          not ALU_LEFT                     when (INST = "010001")else
          SUM                              when (INST = "010010") else
          SUM                              when (INST = "010011") else
          SUM                              when (INST = "010100") else
          SUM                              when (INST = "010101") else
          SUM                              when (INST = "010110") else
          SUM                              when (INST = "010111") else
          ZERO;

result <= result_local;


FC <=
         (not nf and not zf) when  (INST = "010011") else
         zf            when  (INST = "010010") else
         not nf        when  (INST = "010100") else
         nf            when  (INST = "010101") else
         nf or zf      when  (INST = "010110") else
         (not zf)      when  (INST = "010111") else
          zer;
fz_check <= result_local = zero;
FZ <= one when (fz_check) else zer;
FO <= fo_ovf when not(INST = "001001") else zer;  --'0' for SHL

end arch;
```

Appendix B: Multiple Technology Processor (MTP) Specification

This is the Instruction Set Architecture (ISA) view of the MTP. (See Table 2 below.) It summarizes the functional descriptions given in the MTP specification at the assembly language level of behavior. The mnemonic names for the usual arithmetic and logic (ALU) instructions and the data transfer instructions (MV, MVNS, STO, and LD) are given in the first column. (MV and MVNS allow the movement of data through the ALU without modifying it. MVNS disables flag setting on the operation. STO and LD allow data movement between the register file associated with the ALU and off-chip memory.) The third and fourth columns list the range of registers which may be used as data sources for the ALU. The range of registers allowed for storing the result generated by the ALU is shown in the fifth column.

The sixth column shows the effect of each instruction. S1 and S2 are the two inputs to the ALU. D represents the register file offset destination of the ALU output port, and Dbus represents a bus connecting register file and off-chip memory. ADDRD is the destination address specified for the ALU output. FC is a flag set by execution of conditional instructions, a distinction to detect branching. The only effect of conditional instructions is to set this flag. Shift instructions use the double angle-bracket ("<<" or ">>") to denote shifting in the indicated direction. Note that arithmetic shift instructions are further annotated to indicate sign replication in the higher order bits on a shift right (toward lower order bit positions).

The seventh column emphasizes that only the STO and LD instructions use the D Bus.

The last three columns' subjects are three flags: FO, FZ, and FC. FO is set by arithmetic overflow, and FZ is set by an ALU result of zero. Within these last three columns' entries, S indicates that the flag can be modified by execution of the instruction, X indicates that the flag can be set but is not used ("don't care" condition), "-" indicates that the flag cannot be changed by execution of the given instruction, and "0" indicates the known setting.

Mnemonic	Description	source1	source2	(dest.)	Action	DBus Data	FO	FZ	FC
ADD	Addition	R0 - R28	R0 - R28	R0 - R28	D <= S1 + S2	-	S	S	-
ASHL	Arithmetic Shift Left	R0 - R28	R0 - R28	R0 - R28	D <= S1 << Signed	-	S	S	-
ASHR	Arithmetic Shift Right	R0 - R28	R0 - R28	R0 - R28	D <= S1 >> Signed	-	0	S	-
INCR	Increment	R0 - R28	R0 - R28	R0 - R28	D <= S1 + 1	-	S	S	-
DECR	Decrement	R0 - R28	R0 - R28	R0 - R28	D <= S1 - 1	-	S	S	-
NEG	Negation	R0 - R28	R0 - R28	R0 - R28	D <= S1 * -1	-	S	S	-
SHL	Shift Left	R0 - R28	R0 - R28	R0 - R28	D <= S1 <<	-	0	S	-
SHR	Shift Right	R0 - R28	R0 - R28	R0 - R28	D <= S1 >>	-	0	S	-
SUB	Subtraction	R0 - R28	R0 - R28	R0 - R28	D <= S1 - S2	-	S	S	-
Mnemonic	Description	source1	source2	(dest.)	Action	DBus Data	FO	FZ	FC
AND	Boolean AND	R0 - R28	R0 - R28	R0 - R28	D <= S1 and S2	-	X	S	-
NAND	Boolean NAND	R0 - R28	R0 - R28	R0 - R28	D <= S1 Nand S2	-	X	S	-
NOR	Boolean NOR	R0 - R28	R0 - R28	R0 - R28	D <= S1 Nor S2	-	X	S	-
OR	Boolean OR	R0 - R28	R0 - R28	R0 - R28	D <= S1 Or S2	-	X	S	-
XOR	Boolean XOR	R0 - R28	R0 - R28	R0 - R28	D <= S1 Xor S2	-	X	S	-
NOT	Boolean Complement	R0 - R28	R0 - R28	R0 - R28	D <= S1	-	X	S	-
Mnemonic	Description	source1	source2	(dest.)	Action	DBus Data	FO	FZ	FC
EQ	Equality	R0 - R28	R0 - R28	-	FC <= S1 = S2	-	-	-	S
GT	Greater Than	R0 - R28	R0 - R28	-	FC<= S1 > S2	-	-	-	S
GTE	Greater Than Or Equal	R0 - R28	R0 - R28	-	FC <= S1 >= S2	-	-	-	S
LT	Less Than	R0 - R28	R0 - R28	-	FC <= S1 < S2	-	-	-	S
LTE	Less Than Or Equal	R0 - R28	R0 - R28	-	FC <= S1 < = S2	-	-	-	S
NEQ	Not Equal	R0 - R28	R0 - R28	-	FC <= S1 <> S2	-	-	-	S
Mnemonic	Description	source1	source2	(dest.)	Action	DBus Data	FO	FZ	FC
MV	Move Register	R0 - R28	R0 - R28	R0 - R28	D <= S1	-	0	S	-
MVNS	Move No Set	R0 - R28	R0 - R28	R0 - R28	D <= S1	-	-	-	-
NOP	No Operation	none	none	none	none	-	-	-	-
Mnemonic	Description	source1	source2	ADDRD	Action	DBus Data	FO	FZ	FC
STO (=RD)	Read Register	-	-	R0 - R7	DBus <= (ADDRD)	(ADDRD)	-	-	-
LD (=WRT	Load Register	-	-	R0 - R7	(ADDRD)<= DBus	(ADDRD)	-	-	-

Table 2 MTP Instruction Set Architecture